Babel
Users' Guide

TAMARA DAHLGREN THOMAS EPPERLY
GARY KUMFERT JAMES LEEK

Disclaimer

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Release Information

Babel Users' Guide (this document)	UCRL-SM-205559
Babel Source Code (associated software)	UCRL-CODE-2002-054

Babel
Users' Guide

TAMARA DAHLGREN THOMAS EPPERLY
GARY KUMFERT JAMES LEEK

Center For Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, California, USA

June 8, 2005

Preface

This document applies to Babel 0.10.4. It, like the software it documents, is a work in progress.
– The Babel Development Team

Babel in a Nutshell

Babel is a tool that enables software written in different languages to communicate. It accomplishes this task by using an Interface Definition Language (IDL) similar to COM and CORBA. Babel relies on the Scientific Interface Definition Language (SIDL) that is specifically tuned for scientific applications. By expressing software interfaces, or APIs[1], in SIDL the appropriate glue code stubs and skeletons can be generated to facilitate language interoperability. Features unique to SIDL are:

- Dynamic multi-dimensional arrays
- Complex numbers (e.g. $2 + 3i$)
- In-process optimizations
- Special directives for large-scale parallel distributed programming (future)
- Syntax for specifying interface behavior (future)

Babel enables true object-oriented techniques even in non object-oriented languages. The object model that SIDL supports is similar to Java and Objective C where a class can extend at most one class, but implement many interfaces. In C++ speak, an interface is simply a class of all pure-virtual methods. Furthermore, if library developers want object-oriented features but are required to be 100% ANSI C compliant, Babel can meet those constraints. Although the Babel code generator is implemented in Java, the runtime libraries and generated files for C bindings are 100% ANSI C compliant.

Babel can be used as the basis for a component framework, but it is *not* a complete framework by itself. We've added a tiny CCA-compliant framework, called *Decaf*, in our examples/ directory. Decaf demonstrates how Babel can be used to implement a component framework.

SIDL is also a useful communications tool for code development teams since it only expresses the public API. That is, implementation details, which often prove distracting during collaborative design, can be safely avoided by restricting discussions to the interfaces described in SIDL. Furthermore, since SIDL is simple and clean it can be used by Computer Scientists, Math Programmers, and Application Scientists to debate APIs even using only email.

Scope of this Manual

This document is intended as an introduction and tutorial on the use of Babel tools for the generation and use of component software. The Babel tools were designed specifically for scientific applications, therefore most of the examples and exercises here also deal with scientific applications.

This manual assumes the reader is a programmer who is proficient in two or more of the following languages: C, C++, FORTRAN 77, FORTRAN 90, Java, or Python. Furthermore, this manual assumes the reader is familiar with the

[1] Application Programming Interfaces

SPMD[2] programming model that pervades the scientific computing community. Knowledge of and experience with MPI programming is helpful, but not strictly required.

Getting the Software

Babel source is available free of charge on the web. Developed by the Components Project at the Lawrence Livermore National Laboratory Center for Applied Scientific Computing (CASC), it is licensed under the Lesser GNU Public License (LGPL). See the source distribution for details.

The homepage for the Components Project is

http://www.llnl.gov/CASC/components

Conventions

The following typographic conventions are used throughout this manual.

Italic	is used for file and command names. It is also used to highlight comments in examples and to define terms the first time they appear in a document.
`Constant Width`	is used in examples to show the text that is generated, and in regular text to show operators, variables, and the output from commands or programs.
`Constant Slanted`	is used for displaying for SIDL source code. We use a separate font to distinguish SIDL code from generated code.
`Constant Bold`	is used to show user's modifications to generated code and in examples to show user's actual input at a terminal.
Sans Serif Slanted	is used in examples to show variables for which a context-specific substitution should be made. The variable *filename*, for example, would be replaced by the actual filename.

Additionally, we may use specific blocks of text as sidebars to call the readers attention to particular information. Here's one kind.

> **Rationale:** *Often when listing restrictions or requirements, we find it helpful to also explain and document the rationale behind a design decision. In time, the context in which the rationale was based may become irrelevant, making the rationale blocks very useful for understanding when to change a decision.*

We Appreciate Your Feedback

We have tested and verified the information in this manual. Nonetheless, features may have changed or oversights may exist. Please contact us with any issues, corrections, or suggestions for future versions of this manual through snail mail at:

Components Project
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L–365
Livermore, CA 94551

[2] Single Program Multiple Data

or through email to:

`components@llnl.gov`

To find out more about Babel, feel free to subscribe to one or more of the associated distribution lists given below.

- `babel-announce@llnl.gov` is a moderated email forum to which anyone can subscribe (though no-one can post). This is a low-volume alternative for people who want to know about releases and major announcements.
- `babel-dev@llnl.gov` is an open discussion forum about Babel for serious babel users who want to talk about the internal workings of the tools. Anyone can subscribe or send email to this list.
- `babel-users@llnl.gov` is an open discussion forum about Babel for users. Anyone can subscribe or send email to this list.

To subscribe, simply send email to `majordomo@lists.llnl.gov` with the appropriate line(s):

`subscribe babel-announce` *[email-address]*
`subscribe babel-dev` *[email-address]*
`subscribe babel-users` *[email-address]*

where you can explicitly state your email address in *email-address* or, if you leave *email-address* blank, majordomo will use your email ReplyTo: field.

Acknowledgments

Project Alumni: Nathan Dykman, Scott Kohn, and Brent Smolinski

Interns: Melvina Blackgoat, Kirk Kelsey, Sarah Knoop, and Nija Shi

Alpha Testers: Andy Cleary, Jeff Painter, Cal Ribbens

Contributors (Ideas, Bug Reports, Patches, & Code): Rob Armstrong, Ben Allan, Wael Elwasif, Matt Knepley, Boyana Norris, Barry Smith, Jody Winston, and many more.

Sponsors: Babel development originally started as a Strategic Initiative (SI) in the LDRD (Lab Directed R&D) portfolio of Lawrence Livermore National Laboratory.

Current funding is from the DOE/Office of Science SciDAC program as part of the Common Component Technology for Terascale Scientific Simulation (CCTTSS). Also known as the Common Component Architecture.

Software Notices

Babel depends on a great deal of third-party software.

- **JavaCC** is used to generate the SIDL Parser. This is a java.net community project. JavaCC is available under a BSD-style license here: `https://javacc.dev.java.net/`).
- **gnu.getopt** is an implementation of GNU Getopt in Java and is distributed with Babel as a JAR file. It can be downloaded (along with sourcecode) from either the GNU website

  `http://www.gnu.org/software/java/packages.html`

  or the author's website

  `http://www.urbanophile.com/arenn/hacking/download.html.`

The following is the copyright notice for gnu.getopt:

```
/**********************************************************************
/* Getopt.java -- Java port of GNU getopt from glibc 2.0.6
/*
/* Copyright (c) 1987-1997 Free Software Foundation, Inc.
/* Java Port Copyright (c) 1998 by Aaron M. Renn (arenn@urbanophile.com)
/*
/* This program is free software; you can redistribute it and/or modify
/* it under the terms of the GNU Library General Public License as published
/* by  the Free Software Foundation; either version 2 of the License or
/* (at your option) any later version.
/*
/* This program is distributed in the hope that it will be useful, but
/* WITHOUT ANY WARRANTY; without even the implied warranty of
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
/* GNU Library General Public License for more details.
/*
/* You should have received a copy of the GNU Library General Public License
/* along with this program; see the file COPYING.LIB.  If not, write to
/* the Free Software Foundation Inc., 59 Temple Place - Suite 330,
/* Boston, MA  02111-1307 USA
/**********************************************************************/
```

The text for the GNU Library GPL is available at `http://www.gnu.org/copyleft/library.html`.

Contents

Chapter 1

Introduction

Contents

1.1 Babel Facilitates Language Interoperability

Babel was conceived, designed, and built to solve a problem; namely, to make scientific software libraries equally accessible from all of the standard languages. Hence, its goal is language interoperability. The vision goes far beyond calling BLAS[1] implemented in FORTRAN 77 from a C program. At its heart, Babel lets programmers use their tool of choice in developing complete applications using components implemented in one or more distinct programming languages.

For instance, let us say that an application scientist is running a sophisticated C++ code from a Python scripting environment. This can already be easily accomplished with technologies like SWIG. Now let's say that the simulation is showing some erratic behavior and the application scientist wants to extend the `ConvergenceCheck` class to also report some information to a log file. Let's also assume that this application scientist doesn't want to write a new C++ class much less rewrite the current application. What this individual wants to do is derive and utilize a new class in Python from the C++ `ConvergenceCheck` class. Thus, the C++ simulation code will now have to invoke a method on a class implemented in Python, which then dispatches back to the C++ base class after doing its additional logging. This cannot be done in SWIG because SWIG does not support calls from C++ to Python, only from Python to C++. This is an example of a capability that Babel provides that is outside the scope of SWIG.

Figure 1.1 lists many of the primary languages that are of interest to scientific simulation software developers and users. The good news is that there is a path from each language to every other; meaning that calling from one to another is possible. However, the technologies to get from one language to another vary widely, are fraught with pitfalls, and may require calling through a completely different language.

Babel works by providing the technology to define and support the multi-language interoperation of a common subset of functionality through programming language-neutral interface specifications. See Fig. 1.2 to see a graphical representation of the supported languages. It is important to note that this common functionality subset is *far* from a lowest common denominator solution in that Babel actually adds functionality when it is lacking in the host language.

1.2 Scientific Interface Definition Language (SIDL)

In order to support multi-language interoperability, Babel relies on the specification of interfaces in the Scientific Interface Definition Language (SIDL) (pronounced "SIGH-dull"). SIDL is similar to COM and CORBA IDLs, but was designed with an emphasis on scientific computing. Specifically, SIDL currently supports dynamic multi-dimensional arrays and has built-in complex numbers. It will acquire a set of directives to aid in the description of massively parallel distributed objects and additional syntax for specifying interface behavior.

[1] BLAS: Basic Linear Algebra Subroutines

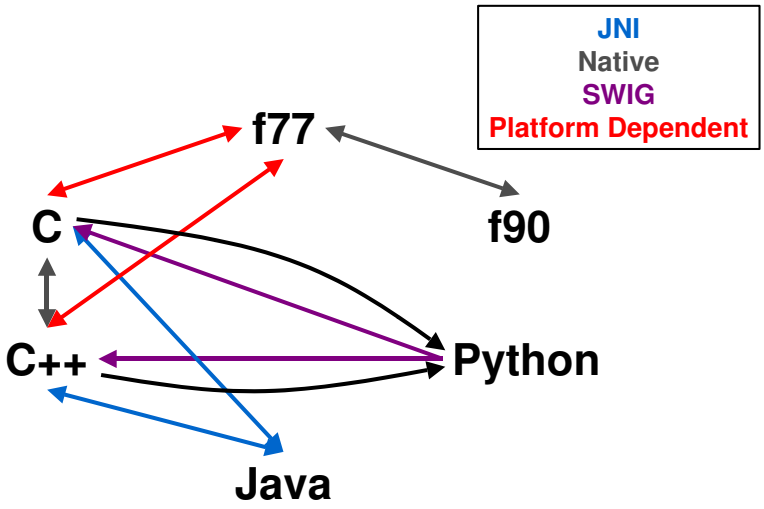

Figure 1.1: Language Interoperability Using Current Technology.

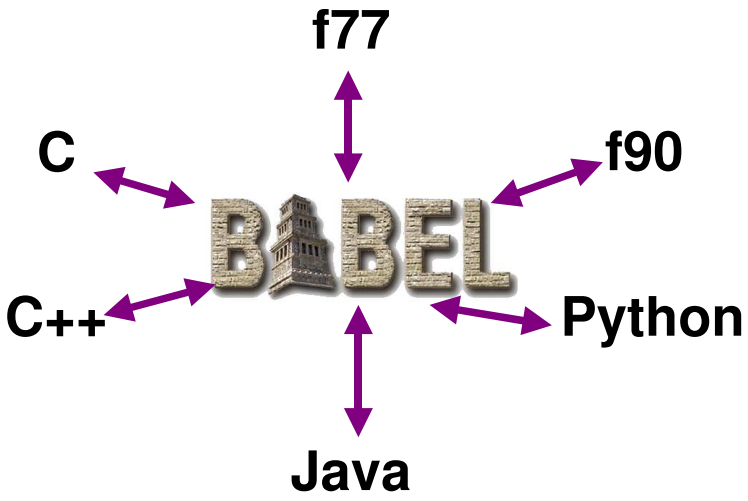

Figure 1.2: Language Interoperability Using Babel.

When it comes to deciding what programming idioms to support across all languages and which ones to reject, SIDL strikes a careful balance between minimalism and completeness. It is *not* a lowest common denominator solution. SIDL is minimal to keep the learning curve as low as possible. It is complete so developers do not feel constrained in how to express their solutions.

SIDL is object-oriented. Its object model closely resembles that of Java and Objective C. In this model there is single inheritance of implementation and multiple inheritance of interfaces. It supports the typical notions of virtual, static, and final methods. SIDL also provides a basic set of features by defining and implementing the basic types for interfaces, classes and exceptions. All types implicitly inherit from these basic types.

The most important concept to grasp about SIDL is that SIDL only defines a public interface that other programs may use to access your code. As a result, all methods defined as part of a SIDL file are public, if you do not want a method to be globally useable, simply do not define it in your SIDL file. Furthermore, all object and class data is implicitly private. There is no way to declare or define data in a SIDL file. Instead, any data required for your code should be declared in the implementation language files. This way, the languages that use your code through Babel may create your objects and pass them around just like any normal piece of data, but they may only access the data through the provided interface.

SIDL also has a complete set of fundamental data types, from booleans to double precision complex numbers. It also supports more sophisticated types such as enumerations, strings, objects, and dynamic multi-dimensional arrays.

SIDL is still a work in progress. Of particular research interest are directives that will be added for parallel distributed object interaction and features to specify behavioral semantics associated with the interfaces.

1.3 Benefits to Customers

Babel has two types of customers: *developer* and *user*. The developer implements a library that will be used by one or more users. Since one goal of the developer is to increase their customer base, the developer writes a SIDL file that effectively publishes the interface to their software in a platform and language neutral manner. The user, on the other hand, may not care or even know that they are interacting with a library through Babel.

Babel provides some features that benefits user and developer alike. The most important aspect to note here is that all Babel objects are reference counted. This feature is critical to encapsulate the memory allocation library (e.g. C's malloc/free or C++'s new/delete) used in the implementation of the object. Users never need concern themselves with when to free up a resource, they only declare when they're done with their reference to that resource. Developers are free to use different memory allocation subsystems in different parts of their code if need be.

1.4 Beyond Babel's Scope

The language interoperability problem is a large one, and though the Babel tools address much of it, there is still a lot that is beyond the scope of our tool. Babel is at its heart a code generator and a runtime library. Consequently, the following features are currently limitations of the Babel tool kit:

Reverse engineering is not supported. That is, there is no support for inspecting or modifying compiled code. In addition, scanning existing software to generate SIDL wrappers is not supported. There are other groups who are pursuing a C++ to SIDL converter. Since SIDL contains different information than what is in a C++ header file, however, such a converter cannot be fully automated without additional help.

Library compatibility is limited. Since Python and Java dynamically load libraries into their virtual machines, using these languages requires the ability to build shared libraries. In general, building shared libraries (particularly from C++) is difficult and error prone. This is compounded by the fact that compiler vendors have no standard way of doing this, and many tools that help building shared libraries don't support C++. One can build a legitimate shared library that still won't work because there are unresolved symbols, or the library was loaded in the wrong mode.

Compiler compatibility is limited. Since the C++ standard does not specify a binary interface and uses a lot of hashing in their symbol tables, there have been no attempts to get libraries from dissimilar C++ compilers to work together. Similarly, although we support FORTRAN 77 and FORTRAN 90, all libraries of Fortran code must be compiled with the same compiler. . . again because of the lack of a standard binary interface.

Despite the aforementioned limitations, Babel does facilitate the development of language interoperable software. However, issues of robust packaging, building, and deployment of language interoperable software still loom on the horizon.

1.5 Summary

Babel consists of a set of tools that are intended to be used for facilitating language interoperability in the scientific computing community. Using interfaces for libraries or components specified in Scientific Interface Definition Language (SIDL) files, Babel can generate corresponding XML representations as well as the source code for the corresponding stubs, intermediate object representations, and implementation skeletons. The generated source code then becomes the foundation for the glue code that is used for language interoperability between callers of libraries and components.

In addition to providing generated code that automatically handles mapping fundamental data type parameters associated with calls between different languages, Babel has built-in support for complex numbers and multi- dimensional arrays. Additional benefits include object reference counting to facilitate memory management.

Finally, Babel's primary goal is to facilitate the development of language interoperable libraries and components. Hence, support for reverse engineering is not provided. Given that Babel has been developed by a research team, there are also limitations associated with shared library and programming language-specific compiler interoperability support that have been looked into but probably will not be addressed in the foreseeable future. Regardless, Babel has proven to be useful to its stakeholders to the point that it is becoming an integral part of the Common Component Architecture (CCA). Refer to papers and presentations on our web site for more information.

1.6 Organization

The remainder of this document is separated into two parts; namely, foundations and supported language bindings. Part I is devoted to describing the SIDL and the Babel tools. It starts with a tutorial to gently introduce the reader to the development of glue code from both the implementation (or server) and user (or client) sides. The following chapter introduces SIDL and Babel basics. Finally, a chapter on advanced topics, such as linking options, is provided.

Part II describes the language bindings currently supported by Babel. At this point, most of the bindings are programming languages. In which case, most have both client- and server-side bindings. However, Babel also supports textual language backends. At this time, Extensible Markup Language (XML) and Scientific Interface Definition Language (SIDL) are the only to textual backends that are supported.

Appendices are included to provide more information on topics such as acronyms, the SIDL Grammar, and SIDL XML. In addition, sections are included that provide advice and tips on troubleshooting.

Part I

Foundations

Chapter 2

Installation

Ideally, Babel will configure and make "out-of-the-box" on most Unix-like machines. If the configuration process detects that certain resources are unavailable, it will correctly disable support for languages or features needing those resources. If this instance of correct behavior is not the intended behavior, then the installer is left to install the external resources and then re-configure, make, and install Babel. This chapter is intended to provide help and reassurance that Babel is indeed configured and installed correctly.

Contents

2.1 Simple Installation

These instructions assume you have a "tarball" (e.g. *.tar.gz file). We have volunteers who put together and manage RedHat RPMs and Debian *.deb distributions of Babel. If you have one of these distros, read their documentation first as it may have details that supersede our own.

A typical build is a simple sequence of

```
% ./configure
# lots of stuff
...
Fortran77 enabled.
C++ enabled.
Java enabled.
Python enabled.
Fortran90 enabled.
% make
# lots more stuff
...
% make install
# not so much stuff
...
```

There are many circumstances where the configuration step will properly terminate with an error, but if the configuration works, the build and installation shouldn't terminate abnormally.

2.1.1 Configure

There are two main choices to be made at configure time: “Where does the software get built?” and “Where does the software get installed?”. The mechanisms for effecting these choices are quite different.

If you want to build software in a separate directory from where the tarball was untarred, this is called a “VPATH build”. VPATH builds are useful if you want to build Babel multiple times with various compilers, flags, or you have a shared filesystem across multiple platforms. It separates the code you generate from things that you were given. The downside is that its more complex to remember where to edit what since original sources will be in the source directory tree and the generated sources and compiled assets will be in the build directory tree.

If you run configure in the directory it appears, (i.e. you typed **`./configure`**) you are performing an “non-VPATH build”. To do a VPATH build, simply cd to the directory you want to be the build directory root, then launch configure from there. The following sequence demonstrates a vpath build

```
% tar zxvf babel-x.x.x.tar.gz
% mkdir babel-linux-build
% cd babel-linux-build
% ../babel-x.x.x/configure
```

Note that the directory where you build Babel should be different from the directory where you install Babel. The default install directory is /usr/local, but can be set to any directory that you have read/write access to. To change the install directory, run configure with the **`--prefix`** option. Since many people do not have root access on their machine (or prefer to install in a local directory when dealing with unfamiliar software), this option is probably the second most heavily used option for configure (first being **`--help`**, which is a good one to try also.)

At the time of this writing (0.9.3), there are two configure scripts in Babel, about 40K lines of shell script each. These configure scripts will then propagate the information they acquire to Makefiles by perform approximately 190 sed substitutions (per Makefile), to the source code by setting approximately 170 preprocessor macros in babel_config.h, and various bits of shell script in the build that do not get propagated to the install directory. The configure script does not modify any source code in Babel’s runtime system or code generator. This means that source code generated by a different Babel installation is usable as long as it gets compiled against the local babel_config.h and linked with the local Babel runtime libraries.

2.1.2 Make

The makefiles are generated by the configure script from Makefile.in templates. The configure script is generated by a tool called autoconf. The Makefile.in’s are generated from Makefile.am files by a separate, but related tool called automake. We also use a tool called libtool to help with libraries. Libtool is written in shell, automake in perl, and autoconf in m4.

After a successful configuration step, if your build fails it is most likely that there is a bug in Babel, autoconf, libtool, or a library of m4 macros from any of the above. It is less likely to be an issue with automake, but possible. Perl and m4 themselves are no longer involved in the process after the configure script is produced, so while there may be a nascent bug in the files they generated, it is unlikely.

2.1.3 Make Check (Optional)

re This is an exhaustive check that can take hours on an average workstation. The number of actual tests run depends on the number of languages that are enabled. In general a driver and an implementation of each test is generated in each enabled language. Then each combination of driver and implementation are run (both statically linked libraries and dynamically loaded libraries, as appropriate) and tested. A test script can actually launch multiple tests, and tests can have multiple parts. At the time of this writing (babel-0.9.3) there are over 13,000 parts tested when all languages are enabled.

2.1.4 Make Install

This transfers built software to the final installation directory. Examples and tests are not installed, nor are Makefiles or dozens of other types of files. Make install also builds javadoc documentation for Babel’s code generator. Since some

libraries are built with install paths in mind, libtool uses a lot of scripts to make things work in their build directory with binaries actually hidden in .lib subdirectories. Make install strips this extra scaffolding away as well.

2.1.5 Make Installcheck (Optional)

This is the same test suite as with make check. The only difference is that it is run against the code in the install directories, not the build directories.

2.2 External Software Requirements

Babel builds on a lot of available software; some optional, some required. Some we ship in our tarball, some we require users to install separately.

2.2.1 Required & Included

- **Java GetOpt:** This is a Java rewrite of GNU GetOpt available at http://www.urbanophile.com/arenn/hacking/download.html. The Babel code generator uses this to parse command line arguments. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.
- **Xerces-J:** Xerces-J is a Java implementation of SAX and DOM XML parsers available from the Apache Software Foundation at http://www.apache.org. The Babel code generator uses this for XML I/O. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.

2.2.2 Required but Separate

- **Unix shell & bintools:** On early 64bit Linux boxes, we found it necessary to rebuild even these basic tools with all 64bit options enabled. Apparently they were originally installed with less attention to detail than necessary. Bintools includes things like cp and mv.
- **C/C++ compiler:** The Babel runtime library and much of the code generated by the Babel code generator will be ANSI C. So that must be available. The C++ compiler should be optional, but at the time of this writing the configure and makefiles didn't reliably support disabling C++.
- **Java:** The Babel code generator is implemented in Java. One can disable the support for Java language bindings, but a working Java would still be needed for just about everything else. We generally stick with Sun's java developer kits (available at http://java.sun.com). Others have run Babel with Kaffe and GJC.
- **libxml2:** This is the Gnome C library for parsing XML files (see http://xmlsoft.org). The Babel runtime library needs version 2.4 or above to parse SCL files for dynamic loading.

2.2.3 Recommended

- **Python:** Needed for the python language binding (obviously) and for the testing harness. Since the Linux kernel is often configured with a Python-based tool, its hard to find a Linux without python already installed. Python can be downloaded from http://www.python.org.

  One important gotcha is a special case where non-python applications create Babel objects implemented in python. In this case, the Babel runtime needs to dynamically load the python virtual machine (libpython.so). Unfortunately, python does not always build a dynamically loadable version of this library by default. If the Babel configure script cannot find a libpython.so, it will disable server-side Python support.

  At the time of this writing, Python cannot be coerced to build a libpython.so on AIX.

- **Numeric Python (NumPy):** This is a scientific array python extension module. It provides native C arrays (and the ability to manipulate very big arrays) similar to python lists. Babel's python language binding requires this extension module available at http://www.pfdubois.com/numpy.

- **Python Meta Widgets (Pmw):** This is a library of GUI widgets built on top of Python's native tcl/tk interface (tkinter). Its available on SourceForge http://pmw.sourceforge.net Pmw is only needed by the GUI in the babel-life supercomputing demo. This Babel implementation of Conway's Game of Life is a separate tarball found in the contrib/ directory of the Babel distro. There is no test for Pmw in Babel's configuration script.
- **Chasm:** Babel uses the Fortran array descriptor library available in Chasm (see http://chasm-interop.sourceforge.net). Chasm is a language interoperability tool in its own right, but as of version 1.0.1, only the array library is considered complete. Without Chasm, the configuration script will disable Fortran 90 support.
- **pthreads:** Needed for Java language binding.

2.2.4 Optional

These packages are used by Babel maintainers in the course of normal development. You'll need these only if you start rewriting code in Babel's distribution.

- **Automake:** Part of GNU Autotools (see http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.
- **Autoconf:** Part of GNU Autotoolssee http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.
- **Libtool:** Part of GNU Autotools (see http://www.gnu.org/software/libtool). Note that we often find need to make minor tweeks to ltmain.sh so a fresh download may generate slightly worse results on some platforms.
- **m4:** Contact us for a patched version that we use (we overflow buffers in the distributed version).
- **JavaCC:** This Java Compiler Compiler is what we use to generate the SIDL parser in Babel. If you are interested in experimenting with changing the SIDL grammar, then edit the compiler/gov/llnl/babel/parsers/sidl/sidl.jj file and rebuilt the parser with this tool. Information available at https://javacc.dev.java.net.
- **LaTeX2HTML:** This is used to generate HTML the HTML version of our manuals.
- **perl:** Needed by automake, LaTeX2HTML and other bits and pieces.

Chapter 3

Basic Babel Code Generation

This chapter describes the Babel code generator and its command line options.
Contents

3.1 Babel is a Compiler

Babel is a compiler. It takes symbols and their interfaces as input and generates either code or a given textual representation. These interfaces may be specified in either Scientific Interface Definition Language (SIDL) or Extensible Markup Language (XML). The form the output takes depends upon the options specified on the command line. Refer to the Section 3.2 for details on command line options. More information on the supported bindings can be found in Part **??** of this document.

3.2 Command Line Options

The entire Babel code generator is written in Java and compiled into a jar file. For convenience, a small script called **`babel`** is provided that *should* set the appropriate environment variables and invoke the Java Virtual Machine on the jar file. To test that the script and jar file are working together properly, simply type **`babel --help`**.

Using Babel

Babel requires exactly one of the following mutually exclusive arguments on the command line.

- **`--help`** : Print options to stdout.
- **`--version`** : Print version of Babel.
- **`--text=`*`form`*** : Generate text equivalent (“sidl” or “xml”) of associated package(s) or generate interface documentation with “html”.
- **`--client=`*`lang`*** : Generate client, or proxy, classes to access library.
- **`--server=`*`lang`*** : Generate the server and client classes to implement the library.
- **`--parse-check`** : Check the SIDL file only.
- **`--generate-sidl-stdlib`** : Regenerate the Babel runtime library.

By far, the three most common uses of Babel will be to generate the Client-side proxies, Server-side implementations, and XML associated with the SIDL file. The last option is essentially used internally when the Babel runtime library is being developed.

Additionally, there are a few supplemental arguments that complete the picture.

- **`--output-directory=`*dir*** : Specifies the root directory associated with the generated files. The default setting is the current working directory.
- **`--generate-subdirs`** : Generates files in a directory tree matching the packaging scope of the SIDL file. This is on by default for languages that have this requirement, such as Java and Python, but off by default for languages that have no such requirement. Hence, code generation for only the latter languages (e.g. C, C++, F77, F90) is effected by this option.
- **`--short-file-name`** : When the `--`generate-subdirs and `--`short-file-names options are used simultaneously, the generated file names will not include package names, just the class or interface symbol. Thus, either long or short names must be used in all clients or servers that have interdependencies; mixing short and long names will result in compile and/or runtime errors.
- **`--repository-path=`*path*** : Specifies a semicolon separated list of directories, or URLs[1] to search for XML Type descriptions. The need for these XML types is to resolve references in the SIDL file. This option can be used multiple times on the same command line. If appropriate, the Babel script adds the default repository path to the command line before dispatching to the Java Virtual Machine.
- **`--no-default-repository`** : Prohibits the use of the default repository in resolving symbols.
- **`--suppress-timestamp`** : Suppresses the insertion of meta-information that could result in generated files that would otherwise not differ from prior executions on the same, unchanged input file. Typically Babel inserts meta-information such as creation time into files it generates. Although this information is useful, it does result in the creation of excessive changes when using version control systems.
- **`--exclude=`*regex*** : This options can be used multiple times. Each time you add a regular expression that will be used to exclude symbols from code generation. No code or XML will be generated for any symbol matching the user provided regular expression. This command line option requires version 1.4.0 or later of the Java runtime environment.
- **`--comment-local-only`** : This option reduces the amount of comments in stub C header files. It will only include the doc comments for locally defined method. It will not include doc comments for inherited methods.
- **`--hide-glue`** : This option causes all non-impl files to be generated in a `glue/` subdirectory. This reduces the "clutter" in the current directory.
- **`--language-subdir`** : This options causes all generated files to be stored in a language-dependent subdirectory; if the **`--generate-subdirs`** option is also used, the language directory will be at the bottom of the hierarchy.
- **`--exclude-external`** : This option causes code to be generated only for the symbols specified on the command line. No code is generated for symbols on which the users symbols depend.
- **`--cxx-ior-exception`** : Earlier versions of the Babel C++ bindings checked the IOR pointer in a given stub before making any calls on it. If the IOR was null, a NullIOException was thrown. It was later found that in certain cases these checks were taking an inordinent amount of time, and since C++ does not normally check pointers before dereferencing them, it was decided that this feature was out of line with the spirit of C++. However, since some code had already been written that used this feature, we could not completely eliminate the checks. Therefore, this command line option was added. Calling babel with it will generate C++ stubs with the checks in them. This option has no effect on other languages.
- **`--vpath=`*dir*** : This option sets the root directory Babel searches first when trying to load implementation files to preserve splicer block contents in the hand edited implementation files. If you are generating server-side C for a concrete class `x.y.z` and you used **`--vpath=/tmp`**, Babel would try to read splicer blocks from `/tmp/x_y_z_Impl.h` and `/tmp/x_y_z_Impl.c`. If it does not find either file in `/tmp`, it also checks the current directory. If you are using **`--generate-subdirs`** with **`--vpath`**, the vpath directory is the root

[1] URLs have colons in them, so this path has to be semi-colon separated, even though UNIX paths are traditionally colon separated.

Table 3.1: Command Line Arguments.

SHORT FORM	LONG FORM	NOTES
-h	--help	Print options to stdout.
-v	--version	Print version of Babel.
-t*form*	--text=*form*	Generate text.
-c*lang*	--client=*lang*	Generate client classes.
-s*lang*	--server=*lang*	Generate server and client classes.
-p	--parse-check	Only check parsing of the SIDL file.
	--generate-sidl-stdlib	Regenerate the Babel runtime library.
-o*dir*	--output-directory=*dir*	Root directory to contain generated files.
-g	--generate-subdirs	Generate sources in directory tree matching SIDL packaging.
-R*path*	--output-directory=*path*	Use specified XML repository(ies) to resolve symbols.
-e*regex*	--exclude=*regex*	Do not generate output for matching symbol(s).
	--no-default-repository	Do not use the default repository to resolve symbols.
	--suppress-timestamp	Suppress time-related metadata generation.
	--comment-local-only	Reduce doc comments in C stub header.
-E	--exclude-external	Do not generate code for dependencies.
-u	--hide-glue	Put glue code in a subdirectory.
-l	--language-subdir	Put code in a language dependent directory.
-x	--cxx-ior-exception	Include Null IOR checks in C++ Stubs.
-V	--vpath	Set the impl (splicer block) root directory.

of the tree, so for the example, Babel would search for `/tmp/x/y/z_Impl.h` and `/tmp/x/y/z_Impl.c`. When appropriate, Babel inserts `#line` directives to refer debuggers to the original file. As its name suggests, this option is useful when making vpath builds using make. Some people also use it to avoid spurious changes to the files managed by their revision control system.

Long and Short Forms

So far, we've shown described the long forms of command line arguments, starting with two hyphens "`--`". There are also short forms for many of the more frequently used commands. See Table **??** for details.

Examples

To create a new XML version of a SIDL file, use the following command:

```
% babel -tXML -omydepot mystuff.sidl
```

To exclude code generation for types whose name begins with "MPI.", use the following command:

```
% babel -sC++ --exclude='^MPI\.'  mystuff.sidl
```

Now suppose a developer wants to implement a library in C++ that corresponds to these types in the SIDL file.

```
% babel -sC++ mystuff.sidl
```

Alternatively, the developer could also create C++ implementation files based on the XML repository. In this case, a list of symbols to be implemented would need to be specified. Assuming that all of the types are in a package called "mystuff", the following command can be issued:

```
% babel -sC++ -Rmydepot mystuff
```

Now suppose a second developer wants to extend this software. A second SIDL file is created then the implementation files in FORTRAN 90 are generated with the following command:

```
% babel -sf90 -Rmydepot newstuff.sidl
```

A user now can download both SIDL files and create their Python bindings to use both libraries with the following command:

```
% babel -cPython -Rhttp://localhost/mystuff/mydepot;
http://www.otherhost.com/newstuff mystuff newstuff
```

Finally, to generate SIDL files for each package based on the XML stored in the repository, the following command is used:

```
% babel -tSIDL -Rhttp://localhost/mystuff/mydepot;
http://www.otherhost.com/newstuff mystuff newstuff
```

Chapter 4

Hello World Tutorial

Contents

4.1 Introduction

This tutorial guides you through the process of writing the classic "Hello World!" example using the Babel tools. In the process, it attempts to teach you how to write a Scientific Interface Definition Language (SIDL) interface description file, generate the library implementation in C++, and write a C main program to call the library. It also illustrates the process for writing a Makefile to compile and link the library and program.

4.2 Writing the SIDL File

The "Hello World!" program will be written in a directory called hello/ and place the client library in a subdirectory hello/lib/:

```
% mkdir hello
% cd hello
% mkdir lib
```

The first step is to write a SIDL file. Recall that SIDL is an interface definition language (IDL) that describes the calling interface for a scientific library. It is used by the Babel tools to generate glue code that hooks together different programming languages. A complete description of SIDL can be found in Chapter **??**.

For this particular application, we will write a SIDL file that contains a class World in a package Hello. Method getMsg() in class World returns a string containing the traditional computer greeting. Using your favorite text editor, create a file called hello.sidl in the hello/ directory containing the following:

```
package Hello version 1.0 {
  class World {
    string getMsg();
  }
}
```

The package statement provides a scope (or namespace) for class World, which contains only one method, getMsg(). The version clause of the statement identifies this as version 1.0 of the Hello package.

4.3 Writing the Implementation

We will write the implementation in the lib/ subdirectory of hello/. The first step is to run the Babel shell script to generate the library implementation code for the SIDL file. We will implement the library in C++. The simplified command to generate the Babel library code (assuming Babel is in your PATH) is [1]:

```
% babel -sC++ -olib ../hello.sidl
```

In this Babel command, the "`-sC++`" flag, or its long form "`--server=C++`", indicates that we wish to generate C++ bindings for an implementation[2]. The "`-olib`" flag, or its long form "`--output-dir=lib`", defines the root directory of where the generated code should be placed.

This command will generate a large number of C and C++ header and source files. It is often surprising to newcomers just how much code is generated by Babel. Rest assured, each file has a purpose and there is a lot of important things being done as efficiently as possible under the hood.

Files are named after the fully-qualified class-name. For instance, a package *`Hello`* and class *`World`* would have a fully qualified name (in SIDL) as *`Hello.World`*. This corresponds to file names beginning with `Hello_World`[3]. For each class, there will be files with `_IOR`, `_skel`, `_stub`, or `_impl` appended after the fully qualified name. *IOR files* are always in ANSI C (source and headers), containing Babel's Intermediate Object Representation. *Impl files* contain the actual implementation, and can be in any language that Babel supports, in this case, they're C++ files. Impl files are the only files that a developer need look at or touch after generating code from the SIDL source. *Skel files* perform translations between the IORs and the Impls. In some cases (like Fortran) the Skels are split into a few files: some in C, some in the Impl language. In the case of C++, the Skels are pure C++ code wrapped in `extern "C" {}` declarations. If the file is neither an IOR, Skel, nor Impl, then it is likely a *Stub*. Stubs are the proxy classes of Babel, performing translations between the caller language and the IOR. Finally, the file babel.make is a Makefile fragment that will simplify writing the Makefile necessary to compile the library. You may ignore the babel.make file if you wish.

The only files that should be modified by the developer (that's you since you're implementing Hello World) are the "Impls", which are in this case files ending with `_Impl.hh` or `_Impl.cc` Babel generates these implementation files as a starting point for developers. These files will contain the implementation of the Hello library. Every implementation file contains many pairs of comment "splicer" lines such as the following:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
    // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```

Any modifications between these splicer lines will be saved after subsequent invocations of the Babel tool. Any changes outside the splicer lines will be lost. This splicer feature was developed to make it easy to do incremental development using Babel. By keeping your edits within the splicer blocks, you can add new methods to the hello.sidl file and rerun Babel without the loss of your previous method implementations. You shouldn't ever need to edit the file outside the splicer blocks.

For our hello application, the implementation is trivial. Add the following return statement between the splicer lines in the `lib/Hello_World_Impl.cc` file:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
```

[1] For information on additional command line options, refer to Section 3.2.

[2] You can also try the "`--help`" flag to list all of the Babel command-line options.

[3] Note: dots are converted to underscores for file naming.

```
    // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
    return std::string("Hello World!");
    // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```

To keep the Makefile simple, we will use some GNU Make features. This Makefile may not work with other make implementations. The GNU gcc and g++ compilers are used in this example. The following Makefile in the lib/ subdirectory will compile the library files and create a shared library named `libhello.so`:

```
# Assumes babel-config is in the current path
.cc.o:
    g++ -fPIC ‘babel-config --includes‘ -c $<
.c.o:
    gcc -fPIC ‘babel-config --includes‘ -c $<

include babel.make
OBJS = ${IMPLSRCS:.cc=.o} ${IORSRCS:.c=.o} \
       ${SKELSRCS:.cc=.o} ${STUBSRCS:.cc=.o}

libhello.so: ${OBJS}
    g++ -shared -o $@ ${OBJS}

clean:
    ${RM} *.o libhello.so
```

You do not necessarily need to create a shared library for this example; you may generate a standard static library (e.g., libhello.a). However, in general, you must generate a shared library if you will be calling your library from Python or Java. To create the shared library archive libhello.so, simply execute make as follows:

```
% cd lib/
% make libhello.so
```

4.4 Writing the Client

We will write the client in the main hello/ subdirectory. The main program will be written in C. File hello.c is as follows:

```
#include <stdio.h>
#include "Hello_World.h"

int main(int argc, char** argv)
{
  Hello_World h = Hello_World__create();
  char* msg = Hello_World_getMsg(h);
  printf("%s\n", msg);
  Hello_World_deleteRef(h);
  free(msg);
}
```

This code creates the Hello_World object, calls the getMsg() method, prints the ubiquitous saying, decrements the reference count for the object, and frees the message string.

There are a few details worth noting here. The C bindings generate function names by combining packages, classes, and method names with underscores (e.g. `Hello_World_getMsg()`. Whenever you see double underscores in Babel generated symbols, they indicate something built-in to (and sometimes specific to) the language binding. The `_create()` method is built-in to every instantiatable class defined in SIDL, triggering the creation of Babel internal data structures as well as the constructor of the actual object implementation.

To generate the C glue code necessary to call the library, we run the Babel tool again, this time specifying C as the target language:

```
% babel --client=C hello.sidl
```

or simply

```
% babel -cC hello.sidl
```

The "`-cC`" flag, or its equivalent long-form "`--client=C`", tells the Babel code generator to create only the C stub calling code, not the entire library implementation. The library libhello.so already contains the necessary IOR, skeleton, and implementation object files. We compile the hello program using the following GNU Make Makefile:

```
.c.o:
    gcc ‘babel-config --includes‘ -Ilib -c $<

include babel.make
OBJS = hello.o ${STUBSRCS:.c=.o}
LIBDIR=‘babel-config --libdir‘
hello: ${OBJS}
    gcc ${OBJS} -o $@ \
      -Wl,-rpath -Wl,lib -Llib -lhello \
      -Wl,-rpath -Wl,$(LIBDIR) -L$(LIBDIR) -lsidl

clean:
    ${RM} *.o hello
```

Note that the "-R" flags tell the dynamic library loader where to find the hello and sidl shared libraries. You could achieve the same behavior through environment variables such as `LD_LIBRARY_PATH`. On some machines and compilers (notably linux-gcc-3.0) the -R flag is no longer supported, so you will have to modify the appropriate environment variable to find the shared library.

Finally, we make the executable and run it:

```
% make hello
% ./hello
Hello World
```

4.5 Final Remarks

Congratulations! You are now ready to develop a parallel scalable linear solver package.

The preceding process may seem to be the most complicated way to write the world's simplest program but, of course, the same process will also work for significantly more complex applications. "Hello World" is small enough to experiment with in the language of your choice. Parallel, multithreaded, scientific simulation codes are another matter entirely.

Chapter 5

SIDL Basics

Contents

5.1 Introduction

This chapter describes the basics of the Scientific Interface Definition Language (SIDL). The goal is to provide sufficient information to enable most library and component developers to begin using SIDL to wrap their software. It begins with an overview of SIDL files followed by an introduction to the fundamental data types. More complex topics such as the object arrays, exceptions, objects, and the XML repository are then addressed.

5.2 SIDL Files

SIDL files are human-readable, language- and platform- independent interface specifications for objects and their methods. SIDL allows you to specify classes, interfaces, and the methods therein. All methods defined in SIDL are public, since the developer is writing them as part of an interface description. Any data you wish a SIDL object to hold is not declared in the SIDL file, and is private. Data should be placed in the implementation skeleton files, and cannot be publicly exported.

Babel reads the SIDL files to generate the appropriate programming language bindings. These bindings, in the form of stub, intermediate object representation (IOR), and implementation skeleton sources, provide the basis for language interoperable software using Babel. In addition, SIDL files are used to populate the XML symbol repository that can serve as an alternate source of interface specifications during the generation of programming language bindings.

Basic Structure

The basic structure of a SIDL file is illustrated below.

```
package <identifier> [version <version>]
{
  interface <identifier> [ <inheritance> ]
  {
    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
      .
      .
      .


    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
  }
```

```
    class <identifier> [ <inheritance> ]
    {
      [<type>] <identifier> (<parameters>) [throws <exception>];
        .
        .
        .


      [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
    }


    package <identifier> [version <version>]
    {
        .
        .
        .
    }
}
```

The main elements are *packages*, *interfaces*, *classes*, *methods*, and *types*. For a more detailed description, refer to Appendix **??**.

Packages provide a mechanism for specifying name space hierarchies. That is, it enables grouping sets of interface and/or class descriptions as well as nested packages. Identified by the *package* keyword, packages have a *scoped* name that consists of one or more identifiers, or name strings, separated by a period (”.”). A package can contain multiple interfaces, classes and nested packages. By default, packages are now re-entrant. In order to make them non-re-entrant, they must be declared as `final`.

Interfaces define a set of methods that a caller can invoke on an object of a class that implements the methods. Multiple inheritance of interfaces is supported, which means an interface or a class can be derived from one or more interfaces.

Classes also define a set of methods that a caller can invoke on an object. A class can extend only one other class but it can implement multiple interfaces. So we have single inheritance of classes and multiple inheritance of interfaces.

Methods define services that are available for invocation by a caller. The signature of the method consists of the return *type*, identifier, arguments, and exceptions. Each parameter has a *type* and a *mode*. The *mode* indicates whether the value of the specified *type* is passed from caller to callee (*in*), from callee to caller (*out*), or both (*inout*). Each exception that a method can *throw* when it detects an error must be listed. These exceptions can be either interfaces or classes so long as they inherit from `sidl.BaseException`. For a default implementation of the exception interfaces, the exception classes should extend `sidl.SIDLException`. Methods and parameter passing modes are discussed in greater detail in Section **??**.

Types are used to constrain the the values of parameters, exceptions, and return values associated with methods. SIDL supports basic types such as `int`, `bool`, and `long` as well as strings, complex numbers, and arrays.

Comments and Doc-Comments

SIDL has the same commenting style as C++/Java and even has a special documentation comment (so called *doc-comment*) similar to those used in Javadoc. One can embed comments anywhere in their SIDL file. Documentation comments should immediately precede the class, interface, or method with which they are associated. Babel replicates documentation comments in the files it generates. It does not replicate plain comments.

```
/*
 *  1. This is a multi-line comment.
 *
 */

//  2. This comment fits entirely on a single line.

/*  3. This comment can fill less than a line. */

/** 4. This is a documentation comment. */

/**
 *  5. Documentation comments can span
 *     multiple lines without the beginning
 *     space-asterisk-space combinations
 *     getting in the way.
 */
```

Consider the above SIDL file fragment.

1. This comment is a regular multi-line comment that is delimited by a slash-star , star-slash ("`/*`", "`*/`") pair.

2. This is a single-line comment that starts with a double slash "`//`" and continues to the end of the line.

3. This comment is the same as # 1 except that it is completely contained on a single line. It can be embedded in the middle of a line anywhere a space naturally occurs.

4. This is a documentation comment. In keeping with Javadoc, Doc++, and other tools, it is delimited by slash-star-star and star-slash ("`/**`", "`*/`") combinations. Documentation comments are important because their contents are preserved by Babel in the corresponding generated files. Doc-comments must directly precede the interface, class, or method that they document.

5. This is a multi-line variant of a doc-comment. Note that initial asterisks on a line are assumed to be for human readers only and are discarded by Babel when it reads in the text. The multi-line doc-comment is the preferred way of documenting SIDL.

Packages and Versions

SIDL has both a packaging and versioning mechanism built in. Packages are essentially named scopes, serving a similar function as Java packages or C++ namespaces. Versions are decimal separated integer values where it is assumed larger numbers imply more recent versions. All classes and interfaces in that package get that same version number. If subpackages are specified, they can have their own version number assigned. If a package is declared without a version, it can only contain other packages. If a package declares interfaces or classes, a version number for that package is required.

```
package mypkg {

}
```

This SIDL file represents the minimum needed for each and every SIDL file. The package statement defines a scope where all classes within the package must reside. Since no version clause is included, the version number defaults to 0.

Packages can be nested. This is shown in the example below. The version numbers assigned to all the types is determined by the package, or subpackage, in which it resides. In the design of the SIDL file, remember that some languages get very long function names from excessively nested packages or excessively long package names.

```
package mypkg version 1.0 {

  package thisIsAReallyLongPackageName {
  }

  package this version 0.6 {
    package is {
      package a {
        package really {
          package deeply version 0.4 {
            package nested {
              package packageName version 0.1 {
              }
            }
          }
        }
      }
    }
  }
}
```

External types can be expressed in one of two ways. The fully scoped external type can be used anywhere in the class description. Alternatively, an *import* statement can be used to put the type in the local package-space. *import* statements can request a specific version of the package, if that version is not found, Babel will print an error. If no version is specified, Babel will take whatever version it is being run on. Babel can not be run on two versions of a given package at the same time, even if you only import or require one of them.

Another way to restrict the package version you use is the *restrict* statement. *restrict* does not import the package, but if you do later import the package or refer to something in that package by it's fully scoped name, Babel will guarantee that the correct version of the package will be used. Also note that all restrict statements must come before the first import statement.

Below is a sample SIDL file, that should help bring all of these concepts together.

```
require pkgC version 2.0; // restrict pkgC to version 2.0, not imported

import pkgA version 1.0; // restrict pkgA version 1.0. Includes class pkgA.A

import pkgB;      // import pkgB regaurdless of version.  Includes class pkgB.B

package mypkg version 2.0 {
  class foo {
    setA( A ); // imported from pkgA, must be pkgA.A-v1.0
    setB( B ); // imported from pkgB, must be pkgB.B, no version restriction
    setC( pkgC.C ); // must be pkgC.C-v2.0
    setD( pkgD.D ); // no version restriction
  }
}
```

Re-entrant Packages

By default, SIDL packages are re-entrant. This means that Babel allows sub-packages to be broken into separate files, but you'd still have to run Babel on all the files at the same time. Here's how it works.

First define the outermost package in a file.

```
package mypkg version 2.0 {

}
```

Table 5.1: SIDL Types

SIDL TYPE	SIZE (BITS)
`bool`	1
`char`	8
`int`	32
`long`	64
`float`	32
`double`	64
`fcomplex`	64
`dcomplex`	128
`opaque`	64
`string`	varies
`enum`	32
`interface`	varies
`class`	varies
`array<Type,Dim>`	varies
`rarray<Type,Dim>` (*index variables*)	varies

Then define a sub-package in a second file.

```
package mypkg.subpkg version 2.0 {

}
```

Note that both files begin with the identical version statement. Now as long as you run Babel on both SIDL files at the same time (with the outermost one first on the commandline), all is fine.

This works because the package statement takes a scoped identifier as an argument. As long as Babel knows that a package `mypkg` exists, it can handle a new package called `subpkg`. (This would also work if `subpkg` were a class. Version statements require an identifier for the outermost package. Since packages cannot have dots "." in their names, the only dots in version statements should appear at the numbers, not the package names.

Running the second file without the first will (and should) generate an error since the enclosing package was not declared. Re-entrance should be used judiciously. This feature may be disabled by labeling a given package as `final`.

5.3 Fundamental Types

Table **??** briefly shows the different data types that are supported in Babel. Refer to each chapter for the language specific bindings for each SIDL type. The "S" in SIDL stands for "Scientific." This emphasis is reflected in the fundamental support for complex numbers (`fcomplex` and `dcomplex`) and dynamic multidimensional arrays (`array<Type,Dim>`).

C++ developers looking at the SIDL syntax for arrays, might think that SIDL is a templated IDL, but this is not so. Although the syntax for SIDL arrays looks like a template, it is specific only to the array type. Developers cannot create templated classes or methods in SIDL.

> **Rationale:** *Although C++ templates are a very powerful programming mechanism, they apply only to C++. For Babel to implement similar hashing routines, method names in languages other than C++ would become prohibitively (thousands of characters) long. Moreover, this C++ template hashing mechanism is compiler specific so while C++ is very good at hiding the expanded template names (unless there is an error to report) we would have to add babel C++ bindings on a compiler by compiler basis.*

Discussion of the various types is broken up into sections. Numeric types such as `bool`, `char`, `int`, `long`, `float`, `double`, `fcomplex`, `dcomplex`, `string`s, as well as information about enumerated types and the opaque type are all covered in this Subsection **??**.

Information about extended types such as Interfaces and Classes along with the methods they contain are described in Section **??**, and Section **??** covers Array.

Numeric Types

The SIDL types *bool*, *char*, *int*, *long*, *float*, *double*, *fcomplex*, and *dcomplex* are the smallest and easiest data types to transfer between languages transparently. They all have a fixed size and can just as reasonably be copied as passed by reference.

Most languages natively support all of these data types (though perhaps less so with complex types). There are a few notable exceptions that may be of interest.

ANSI C does not define the size of `int` and `long`, only that the latter be at least as big as the former. As of the C99 standard, there are types `int32_t` and `int64_t` that are signed integers that explicitly support a fixed number of bits. Most compilers already have these symbols defined appropriately in `sys/types.h` (pre C99 standard) or `inttypes.h`.

Python defines its `int` and `long` to be equivalent to C, and therefore suffers the same platform dependent integer size problem with less flexibility for a workaround. It is not uncommon for regression tests involving longs and Python to fail on certain platforms. Python 2.2 has a patch to make SIDL long support better.

Strings

Strings are an interesting datatype because they are fundamental to many pieces of software, but represented differently by practically every single programming language. Strings can have a high overhead to support language interoperability because there is invariably so much copying involved.

FORTRAN 77 and 90 support for strings is limited to a predetermined buffer size. Since the results of a string assignment into that buffer in FORTRAN does not propagate the length of the string, trailing whitespace is always trimmed for any string begin passed out from a FORTRAN implementation.

Opaque

The *opaque* type is dangerous and rarely useful. However, there are particular times when an opaque type is the only way to solve a problem; for example, it is one of the few portable ways to implement an object with state in Fortran 77 (see Section **??**). When a SIDL file uses an *opaque* type, Babel guarantees only bits will be relayed exactly between caller and callee. If there is a need to pass more information than an opaque provides, than the developer can simply pass a pointer to that information.

Use of a *opaque* carries a heavy penalty. When Babel matures enough to support distributed computing, any method calls with *opaque* in the argument list (or return type) will be restricted to in-process calls only.

> **Rationale:** *Since* `opaque` *is typically used for a pointer to memory, this sequence of bits has no meaning outside of its own process space.*

Enumerations

An enumeration is typically used in programming languages to specify a limited range of states to enable dealing with them by names instead of hard-coded values. For language interoperability purposes — especially to support this concept on languages with no native support — we've had to create specific rules for the integer values associated with enumerated types.

```
package enumSample version 1.0 {

  // undefined integer values
  enum color {
    red, orange, yellow, green, blue, violet
  };

  // completely defined integer values
  enum car {
```

```
    /**
     * A sports car.
     */
    porsche = 911,
    /**
     * A family car.
     */
    ford = 150,
    /**
     * A luxury car.
     */
    mercedes = 550
  };

  // partially defined integer value
  enum number {
    notZero,// This non-doc comment will not be retained.
    notOne,
    zero=0,
    one=1,
    negOne=-1,
    notNeg
  };
}
```

Above is a sample of enumerations taken directly from our regression tests. It defines a package `enumSample` that contains three enumerations. C/C++ developers will find the syntax very familiar. When defining an enumeration, the actual integer values assigned can be undefined, completely defined, or partially defined.

SIDL defines the following rules for adding integer values to enumerated states that don't have a value explicitly defined.

1. Error if two states are explicitly assigned the same value
2. Assign all explicit values to their named state.
3. Assign smallest unused non-negative value to first unassigned state in enumeration.
4. Repeat 3 until all states have assigned (unique) values.

To verify the application of these rules, the `enumSample.number` enumeration will have the following values assigned to its states: `NotZero`=2, `NotOne`=3, `zero`=0; `one`=1, `negOne`=-1, `notNeg`=4.

5.4 Arrays

One of the features that separates SIDL and BABEL from Microsoft's COM/DCOM and the OMG's CORBA is support for multi-dimensional arrays. SIDL is designed to serve the high performance computing community, so we anticipate that both SIDL object developers and object clients may require direct access to the underlying array data structure to try to optimize instruction pipelining or cache performance. The purpose of this document is to describe the functional API to the SIDL array data structure and the underlying data structures. This presentation will focus on the C API for arrays because it is the basis for the other language APIs, so they will likely reflect its idiosyncrasies.

There are two main kinds of arrays in SIDL: normal arrays and r-arrays. R-arrays are a specialized form of array for numeric types that has a simpler interface from C, C++, FORTRAN 77 and FOTRAN 90. Normal arrays are used for all SIDL types.

The SIDL array API and data structure can be used in client code to prepare argument for passing to a SIDL method, and it is used inside the implementation code to get data and meta-data from incoming arguments.

Normal SIDL arrays can be "row-major" or "column-major". They are not parallel array classes, and not particularly sophisticated, but they are very, very general. These are meant to generalize the array types built into many languages, not to provide a general array component that everyone will use. It is expected for parallel array libraries to build on top of the array type presented into SIDL.

R-arrays

There are two kinds of SIDL arrays: normal SIDL arrays and raw SIDL arrays called r-arrays. Normal SIDL arrays provide all the features of a normal SIDL type. They can be passed as `in`, `inout`, or `out` parameters, and they can be returned as a method return value. Normal SIDL arrays can be allocated or borrowed, and they are reference counted. You can also pass `NULL` as a normal SIDL array.

SIDL r-arrays exist to provide a lower access to numeric arrays from C, C++, Fortran 77, Fortran 90 and future languages as appropriate. For example, a one-dimensional r-array in C appears as a double pointer and a length parameter. To highlight the contrast, normal SIDL arrays appear as a struct in C, a template class in C++, an 64-bit integer in Fortran 77 and a derived type in Fortran 90.

R-arrays are have more restrictions in how they can be used. Here is how r-arrays are more constrainted:

1. Only the `in` and `inout` parameter modes are available for r-arrays. R-arrays cannot be used as return values or as `out` parameters.
2. R-arrays must be contiguous in memory, and multi-dimensional arrays must be in column-major order (i.e., Fortran order).
3. `NULL` is not an allowable value for an r-array parameter.
4. The semantics for `inout` r-array parameters are different. The implementation is not allowed to deallocate the array and return a new r-array. `inout` means that the array data is transferred from caller to callee at the start of a method invocation and from callee to caller at the end of the a method invocation.
5. The implementation of a method taking an r-array parameter cannot change the shape of the array.
6. The lower index is always 0, and the upper index is $n - 1$ where n is the length in a particular dimension. This is contrary to the normal convention for Fortran arrays.
7. It can only be used for arrays of SIDL `int`, `long`, `float`, `double`, `fcomplex`, and `dcomplex` types.

Rationale: *The way r-arrays are passed to the server-side code, particularly Fortran 77, makes it impossible for them to be allocated or deallocated. This makes* `out` *and return values impossible. Because the data has to be accessible directly from Fortran 77 without any additional meta-data, the array data must be in column-major order.*

Arrays of char are not currently supported for r-arrays because in some languages characters are treated as 16-bit Unicode characters.

The advantages of r-arrays include:

- Arrays appear more “natural” in C, C++, Fortran 77, Fortran 90 and future low level languages.
- Developers need less or no code to tranlate between their array data structure and SIDL’s array data structure.
- SIDL generated APIs can have signatures very similar if not identical to well known legacy APIs.
- Less performance overhead because r-arrays can avoid a call to `malloc` and `free`.

When you declare an r-array, you also declare the index variables that will hold the size of the array in each dimension. For example, here is an method to solve one of the fundamental problems of linear algebra, $Ax = b$:

```
void solve(in    rarray<double,2> A(m,n),
           inout rarray<double>   x(n),
           in    rarray<double>   b(m),
           in    int              m,
           in    int              n);
```

In this example, `A` is a 2-D array of doubles with `m` rows and `n` columns. `x` is a 1-D array of doubles of length `n`, and `b` is a 1-D array of doubles of length `m`. Note that by explicitly declaring the index variables, SIDL takes avoid using extra array size parameters by taking advantage of the fact that the sizes of `A`, `x` and `b` are all inter-related. The explicit declaration also allows the developer to control where the index parameters appear in the argument list. In many cases, the argument types and order can match existing APIs.

The mapping for the solve method will be shown for C, C++, Fortran 77 and Fortran 90 in the following chapters. In languages that do not support low level access such as Python and Java, r-arrays are treated just like normal SIDL arrays, and the redundant index arguments are dropped from the argument list. The indexing information is available from the SIDL array data structure.

SIDL Language Features

As of release 0.6.5, interface definitions can specify that an array argument or return value must have a particular ordering for a method. The type `array<int, 2, row-major>` indicates a dense, [1] two-dimensional array of 32 bit integers in row-major order; and likewise, the type `array<int, 2, column-major>` indicates an dense array in column-major order. Some numerical routines can only provide high performance with a particular type of array. The ordering is part of the interface definition to give clients the information they need to use the underlying code efficiently. The ordering specification is optional.

For one-dimensional arrays, specifying `row-major` or `column-major` allows you to specify that the array must be dense, that is stride 1. Otherwise, for one-dimensional arrays row-major and column-major are identical.

If you pass an array into a method and the array does not have the specified ordering, the skeleton code will make a copy of the array with the required ordering and pass the copy to the method. This copying is necessary for correctness, but it will cause a decrease in performance. The implementor of the method can count on an incoming array to have the required ordering.

For `out` parameters and return values, an ordering specification means that the method promises to return an array with the specified ordering. The implementation should create the `out` arrays with the proper ordering; because if it does not, the skeleton code will have to copy the outgoing array into a new array with the required ordering.

For `inout` parameters, an ordering specification means the ordering specification will be enforced by the skeleton code for the incoming and outgoing array value.

At the time of writing this, the ordering constraints are enforced for Python implementation because Python uses Numeric Python arrays, so BABEL cannot control the array ordering as fully. The Python skeletons do force outgoing arrays (i.e., arrays passed back from Python) to have the required ordering.

Independent and borrowed arrays

From a memory perspective, there are two main kinds of arrays: independent and borrowed. The independent arrays owns and manages its data. It allocates space for the array elements when the array is created, and it deallocates that space when the array is finally destroyed.

The borrowed array does not own or manage its data. It borrows its array element data from another source that it cannot manage, and it only allocates space for the index bounds and stride information. The rationale for borrowed arrays is to allow data from another source to temporarily appear as a SIDL array without requiring data be copied.

If you `slice` an independent array, the resulting array is also considered independent even though it borrows data from the original independent array. The resulting array can still manage its data by retaining a reference to the original array; hence, its element data cannot disappear until the resulting array is destroyed. If you `slice` a borrowed array, the resulting array is also borrowed because like its original array, it doesn't manage the underlying data.

In the Babel generated code, r-arrays are converted to borrowed arrays. These borrowed arrays are allocated on the stack rather than on the heap to improve performance of r-arrays.

The Life of an Array

The existence of borrowed arrays causes the arrays to deviate from the normal reference counting pattern. You may recall that all arrays are reference counted, and an array's resources are reclaimed when the reference count goes to zero. However, a borrowed array's array element data will disappear whenever the source of the borrowed data determines that it should regardless of the reference count in corresponding the SIDL array. This behavior means that developers should consider any SIDL array that they did not create themselves, for example incoming arguments to methods, as potential borrowed arrays. When a method wants to keep a copy of an array that might be a borrowed array, it should use the `smartCopy` method documented below.

Here are some rules of thumb about the use of borrowed arrays:

[1] meaning non-strided